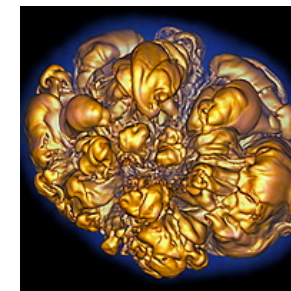
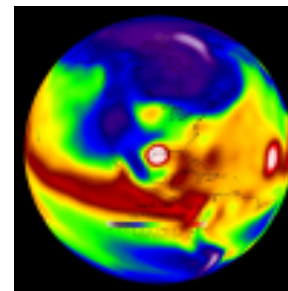
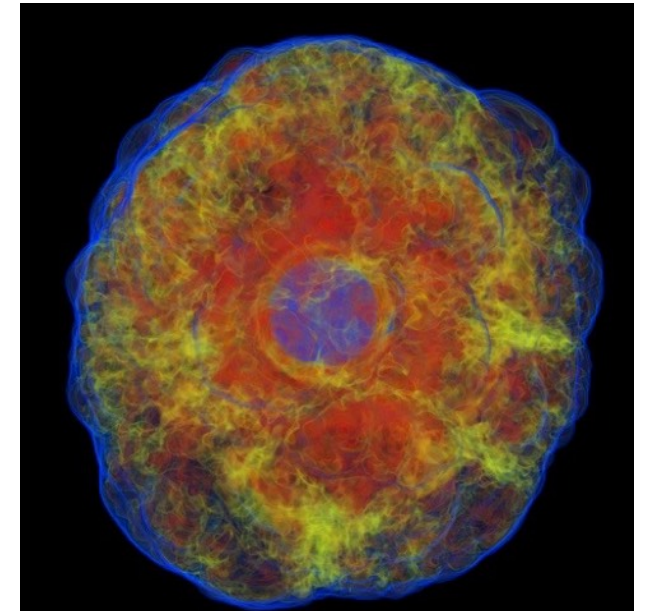
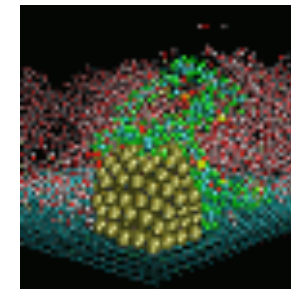
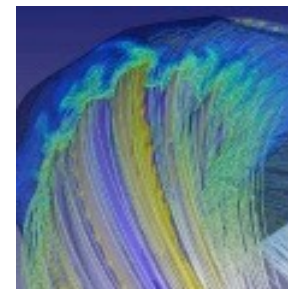
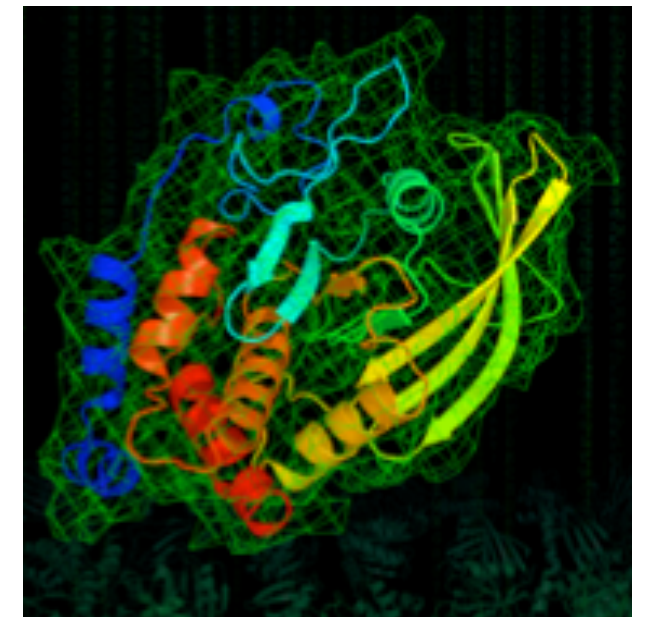
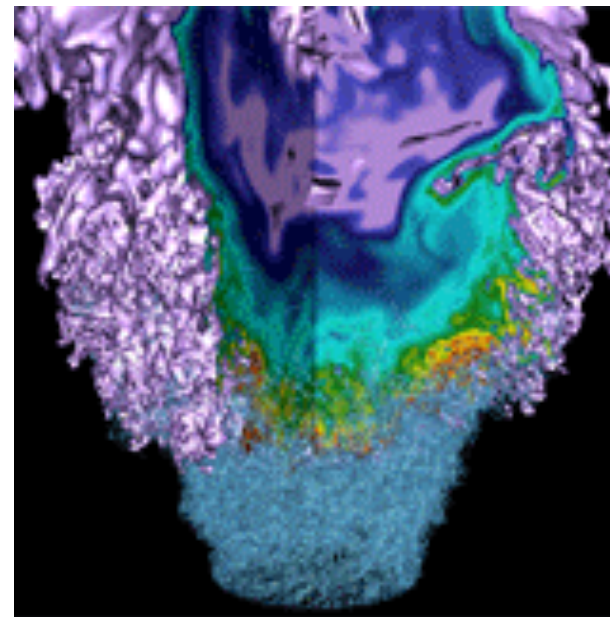


Optimizing Code for Intel Xeon Phi 7250 (Knight's Landing)



Thorsten Kurth

NUG Training, 06/09/2017

Multicore vs. manycore



- **multicore (Edison)**

- 5000 nodes
- 12 physical cores/CPU
- 24 HW threads/CPU
- 2.4-3.2 GHz
- 4 DP ops/cycle
- 30 MB L3 cache
- 64 GB/node
- 100 GB/s memory bandwidth

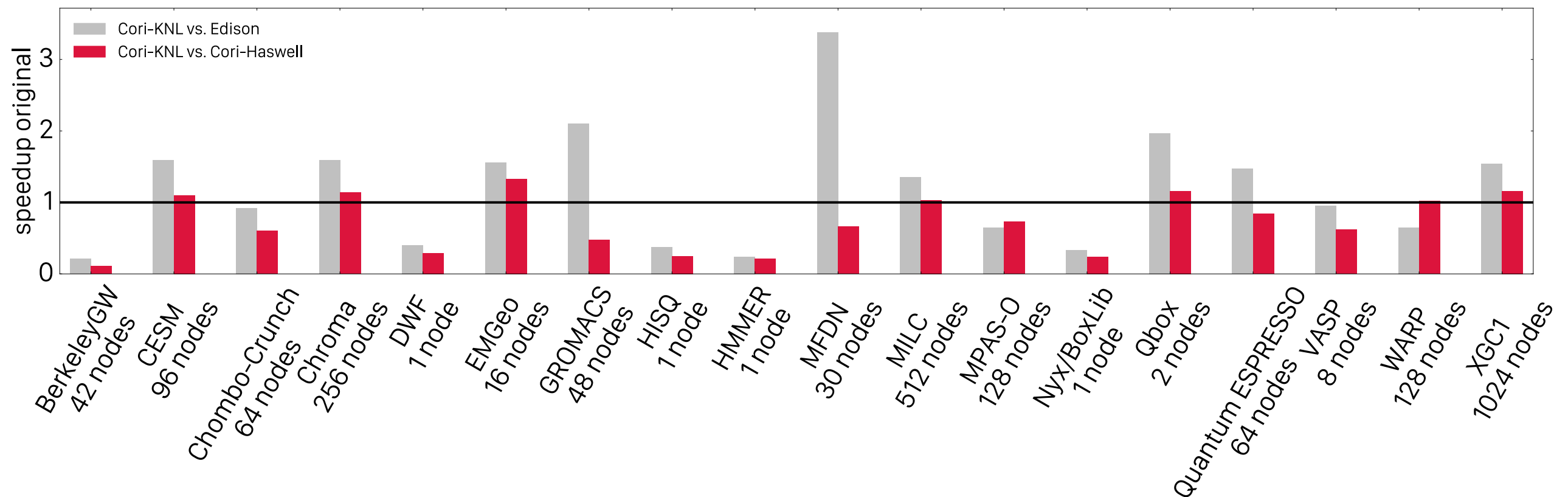
- **manycore (Cori-KNL)**

- 9600 nodes
- 68 physical cores/CPU
- 272 HW threads/CPU
- 1.2-1.6 GHz
- 2x8 DP ops/cycle
- no L3 cache
- 16 GB/node (fast)
96 GB/node (slow)
- 450 GB/s memory bandwidth (fast)

Recompile and go?



- **x86-64 compatible:** can use codes for older architectures or recompile
- **self-hosted:** no need for offloading



- median speedup vs. Edison: 1.15x
- median speedup vs. Haswell: 0.70x

Why should I optimize my code?



- pros
 - **get more for your bucks:** making efficient use of existing manycore HPC systems
 - **fast success possible:** many low hanging fruits in unoptimized codes
 - **investing in the future:** heterogeneous architectures are energy efficient and thus will stay around for a while
 - **benefits on multicore:** optimizations targeting manycore architectures mostly improve performance on multicore systems as well
- cons
 - **effort:** many most beneficial optimizations require significant code changes
 - **investing in the future:** what if I bet on the wrong horse?

Optimization targets



- **single node performance**
 - start here: for representative local problem size, single node performance is upper bound of what you get in multi-node
 - many optimization opportunities, fast turnaround times
 - many profiling tools available
- **multi-node performance**
 - fewer optimization opportunities, profiling/debugging tedious
- **IO performance**
 - not many opportunities for improvement

Where do I start?



- **get to know your application:** don't assume you already do!
- **determine hotspots**
 - **manual timers:** be careful with thread safety/sync barriers
 - **profiling tools:** NERSC offers a lot
 - [CrayPat](#) (very lightweight)
 - [Advisor](#) (find time-consuming loops)
 - [VTune](#) (can do a lot of things but also very slow)
 - [MAP](#) (comparably lightweight)
- **found hotspots, now what?**

What architectural feature shall I target?



- KNL has many new features to explore
 - many threads
 - bigger vector units
 - complex intrinsics (ISA)
 - multiple memory tiers
- **understand your hotspots**
 - **compute bound:** more threads, vectorization, ISA
 - **memory BW bound:** memory tiers, more threads
 - **memory latency bound:** more threads, vectorization

Prerequisites - compile and run



- **recompile your code for KNL:** code for older CPUs is supported but those do not make full use of new architecture
 - **Cray (wrappers):**
`module swap craype-haswell craype-mic-knl`
 - **Intel:** `-xmic-avx512`
 - **GNU:** `-march=knl`
- **use proper OpenMP settings:**
`export OMP_NUM_THREADS=64`
`export OMP_PLACES=threads`
`export OMP_PROC_BIND=spread`
- use [job-script-generator](#) on [my.nersc.gov](#) or [NERSC website](#)
- **node configuration:** use `-C knl,quad,cache` as a start

Prerequisites - #FLOPS



- #FLOPS: number of floating point operations
 - manual calculation:
 - float addition and multiplication: +1
 - complex multiplication: +6 (4 multiplications+2 additions)
 - etc.

- [measure with SDE](#):

```
1  __SSC_MARK(0x111); // start SDE tracing, note it uses 2 underscores
2  -
3  ▼  for (k=0; k<NTIMES; k++) {
4      <my super expensive loop body>
5  ▲  }
6  -
7  __SSC_MARK(0x222); // stop SDE tracing|
```

```
1  srun -n ... sde64 -knl -d -iform 1 -
2      ..... -omix my_mix.out -i -global_region -
3      ..... -start_ssc_mark 111:repeat -stop_ssc_mark 222:repeat -
4      ..... | - my_super_slow_app.exe
```

- using SDE is more precise, because it accounts for masking

Prerequisites - #BYTES



- **#BYTES: number of bytes transferred from main memory**
 - manual calculation (**not recommended, but good check**):
 - count the bytes of data to be read and written in the kernel
 - **does not account for data reuse through caching**

- [measure with VTune](#):

```
1  #include <ittnotify.h>
2
3  __itt_resume(); // start VTune collection, again use 2 underscores
4
5  for (k=0; k<NTIMES; k++) {
6      <my super expensive loop body>
7  }
8
9  __itt_pause(); // stop VTune collection
```



```
1  srun -n ...
2      .... amplxe-cl
3      .... -start-paused
4      .... -r my_vtune
5      .... -collect memory-access
6      .... -no-auto-finalize -trace-mpi
7      .... -- my_super_slow_app.exe
```

- precisely obtain uncore counter events

What is limiting my performance?



- Roofline performance model

- arithmetic intensity

$$AI = \frac{\#FLOPS}{\#BYTES}$$

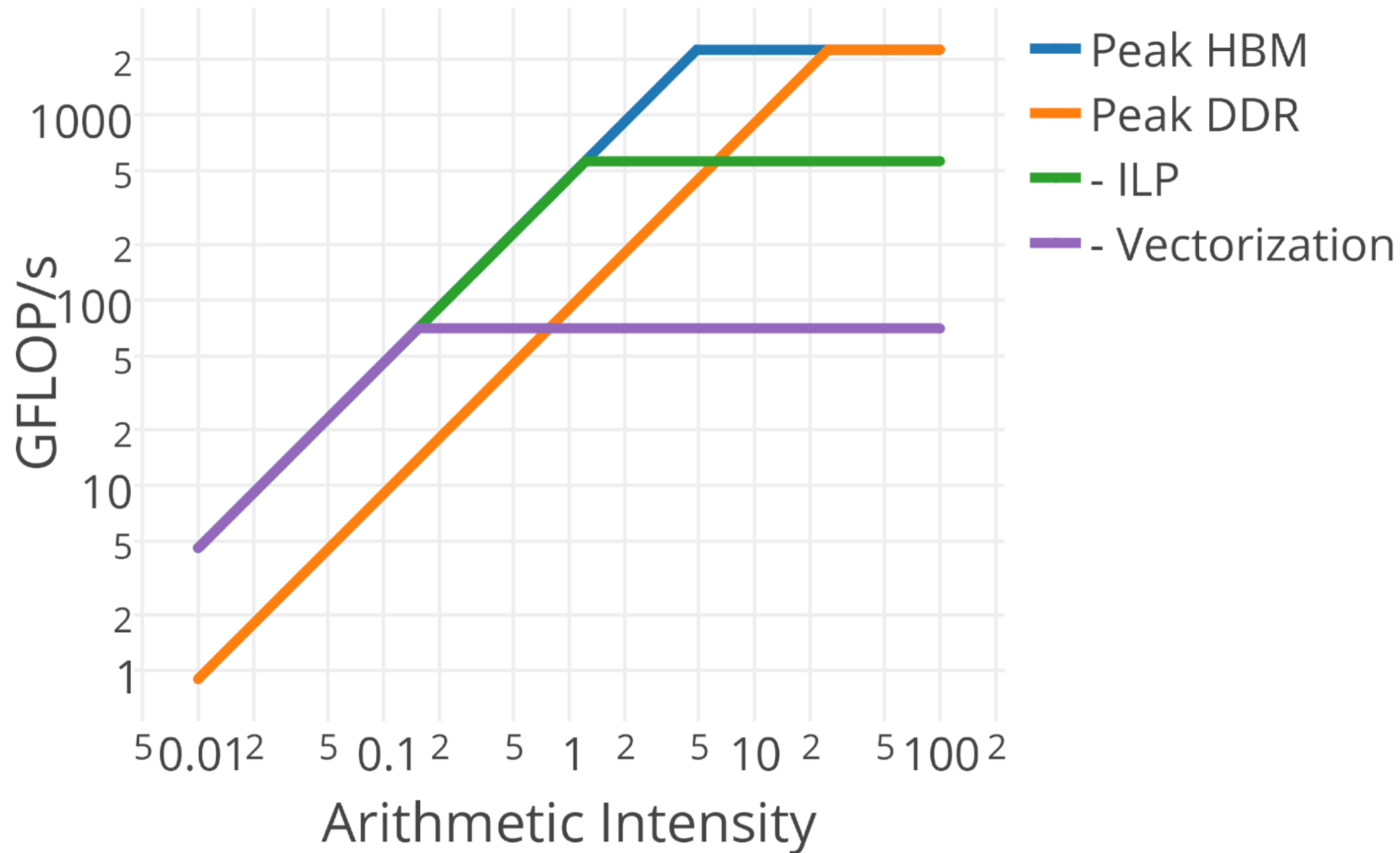
- performance

$$P = \frac{\#FLOPS}{\text{time}[s]}$$

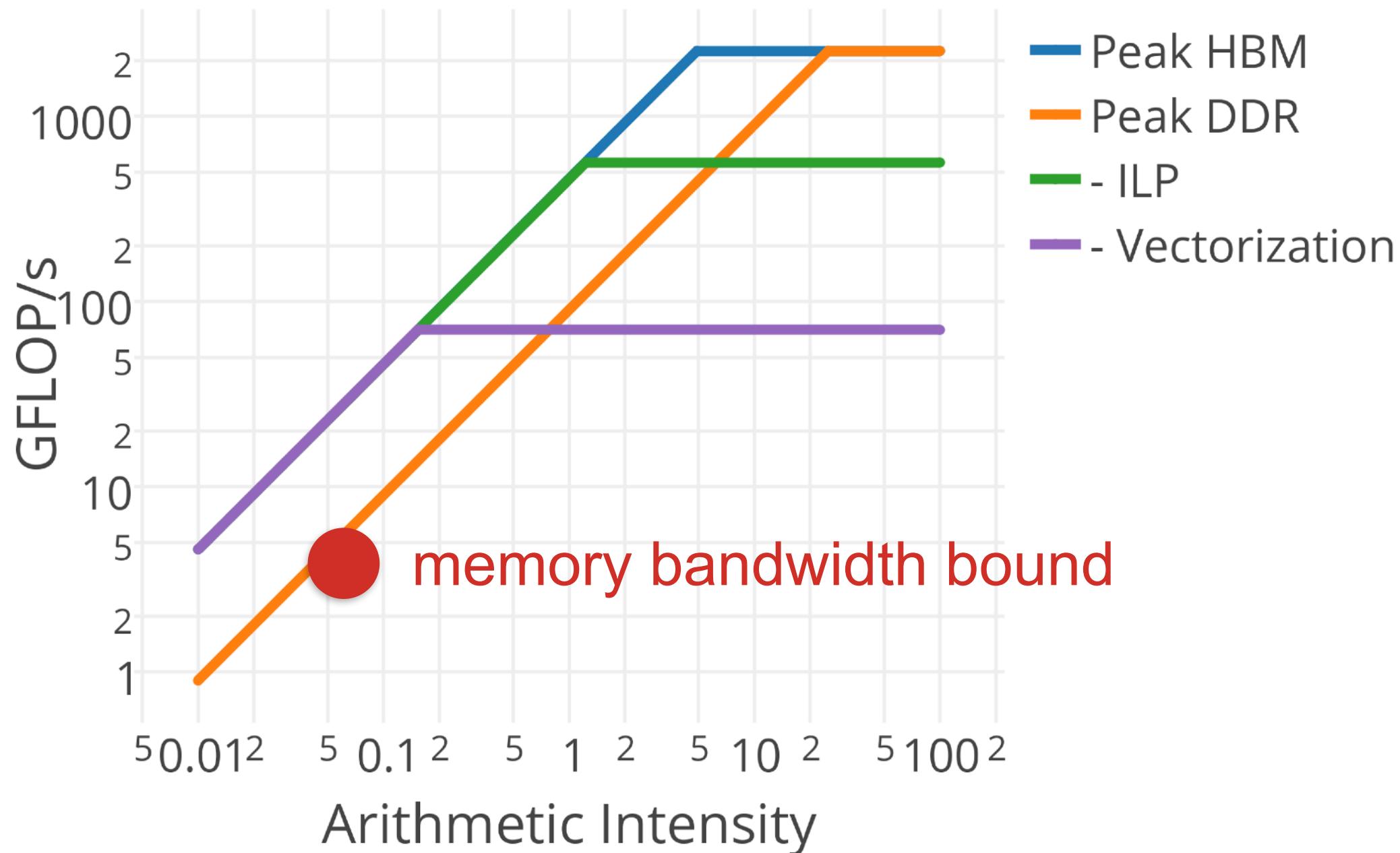
- plot P vs. AI with architectural roofline R

$$R(AI) = \min(\text{memory_bw} \cdot AI, \text{peak_flops})$$

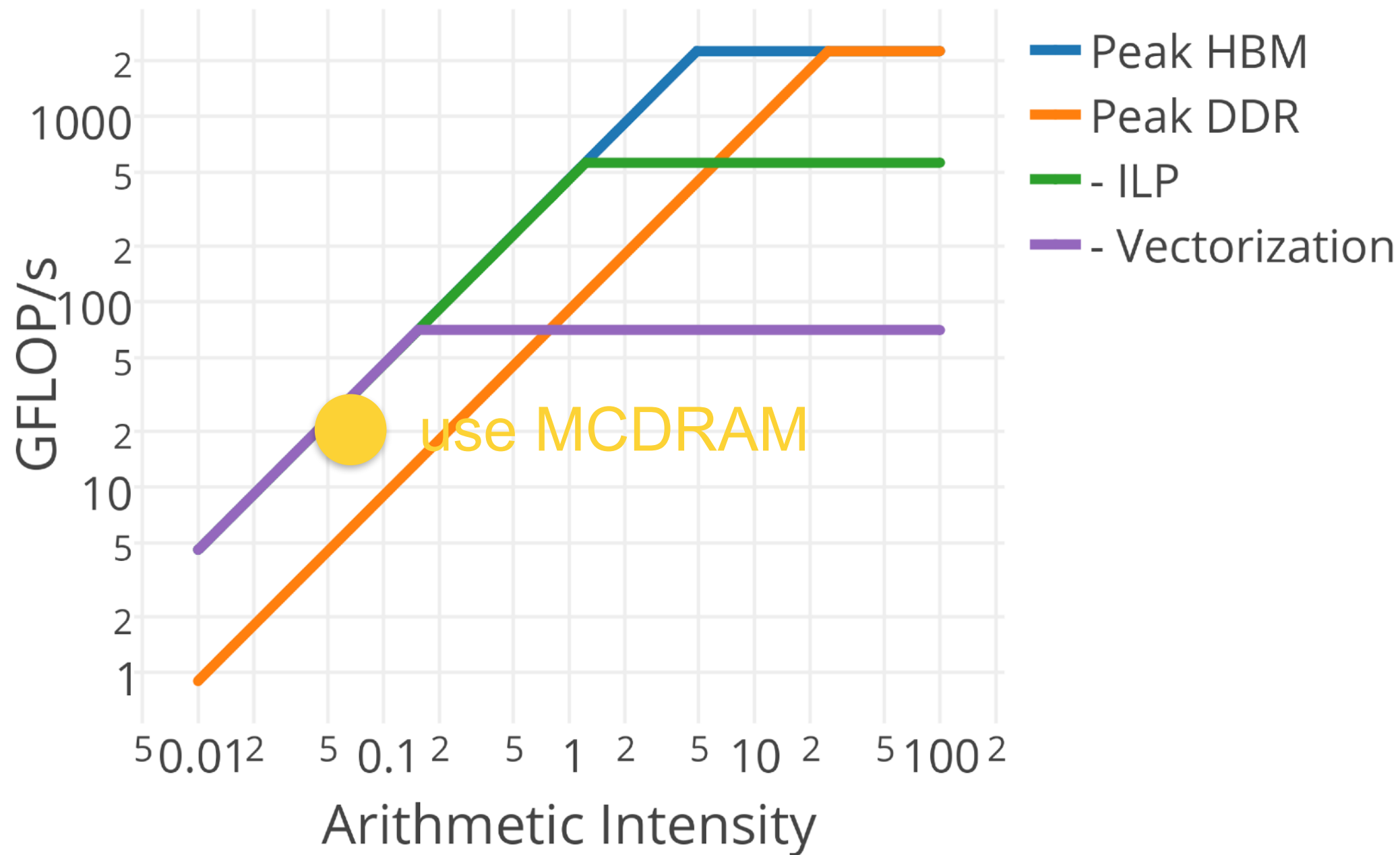
Example roofline



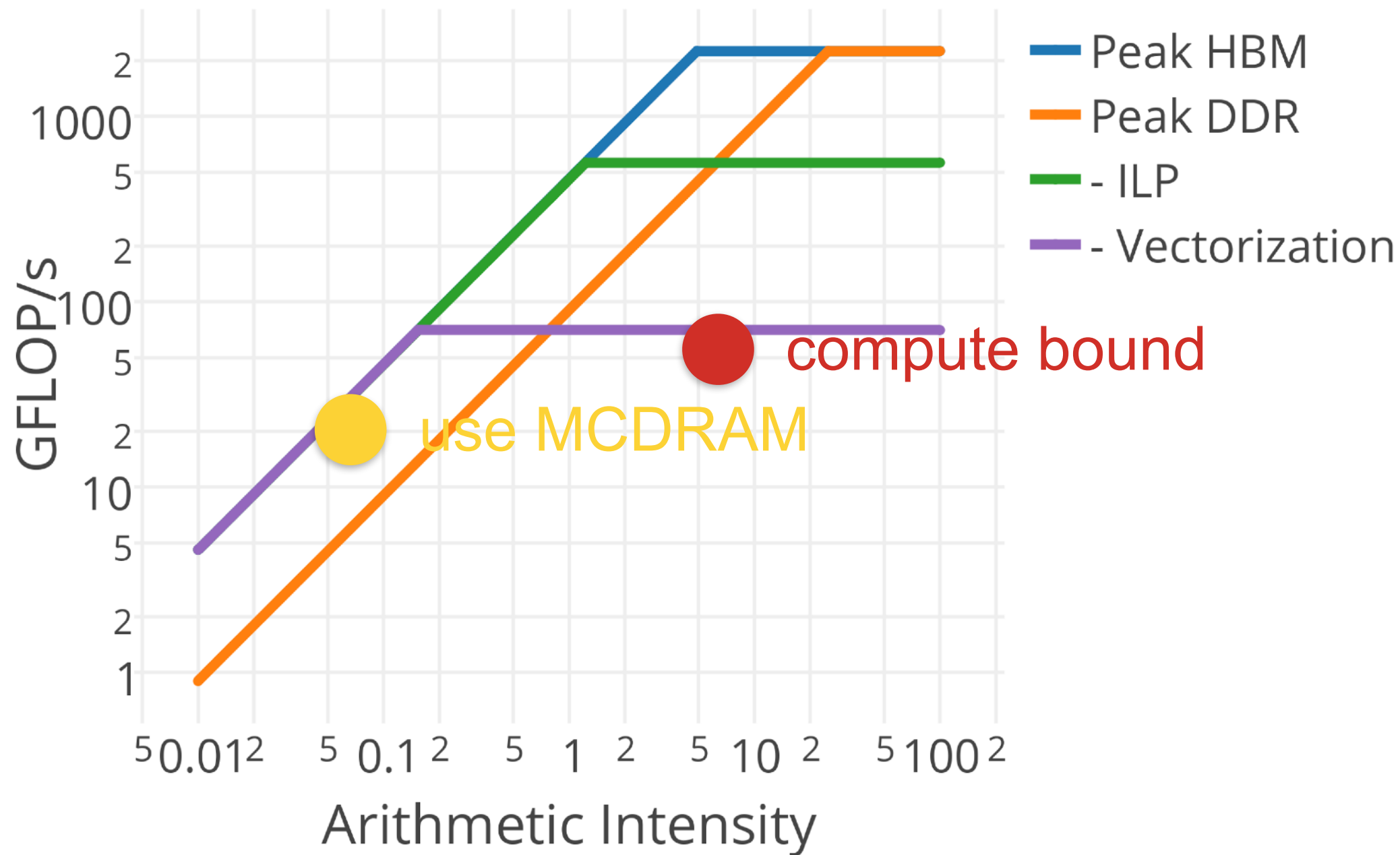
Example roofline



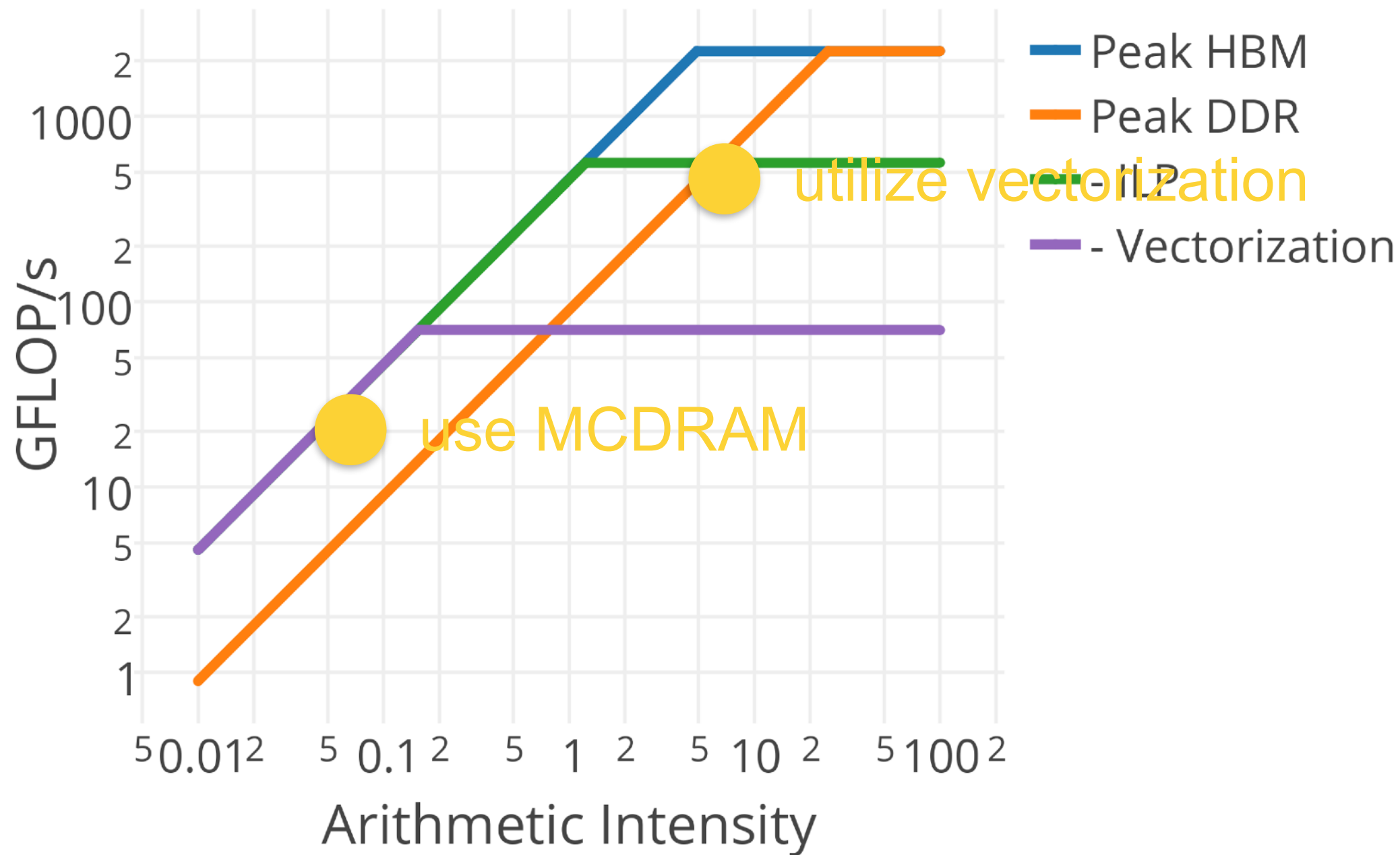
Example roofline



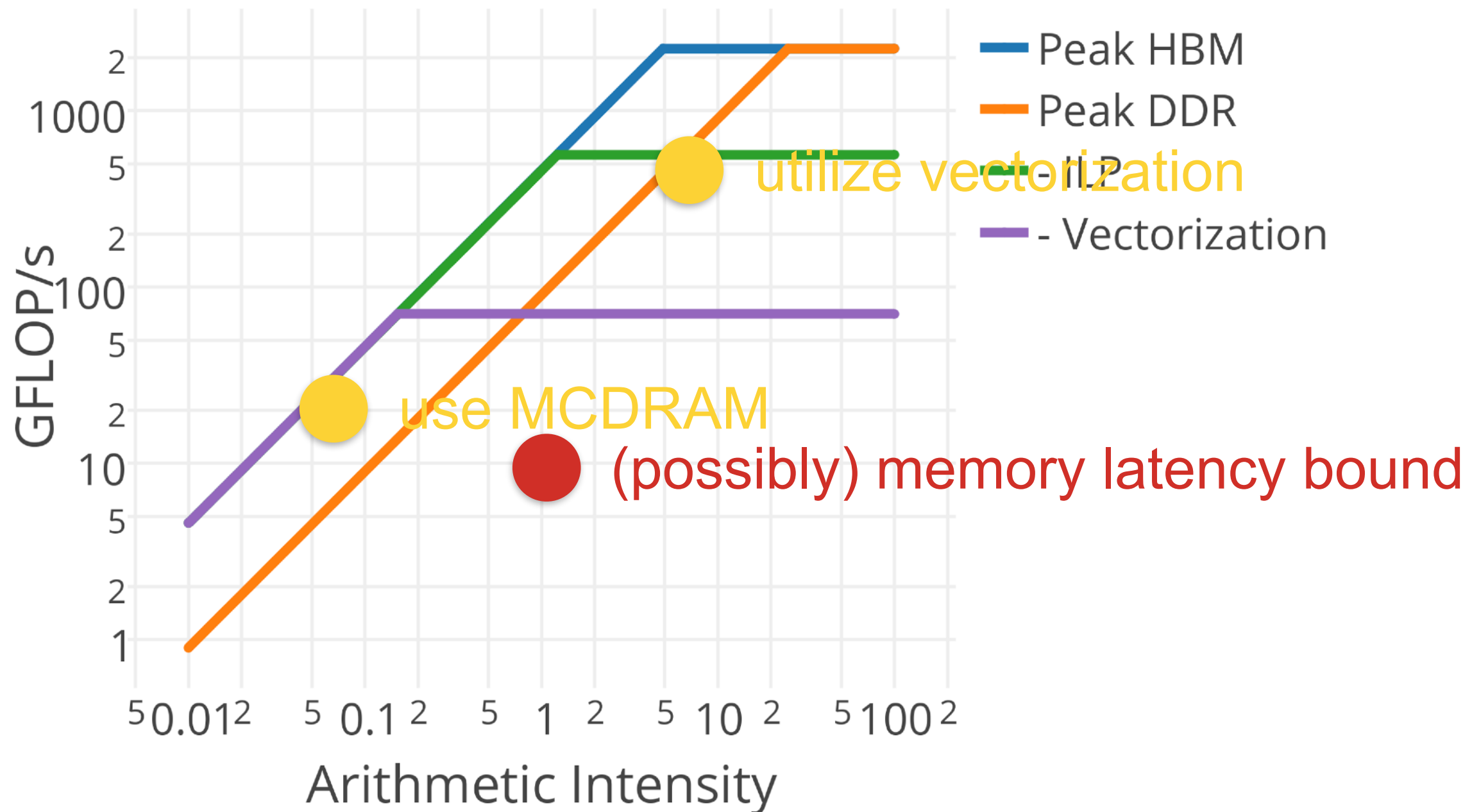
Example roofline



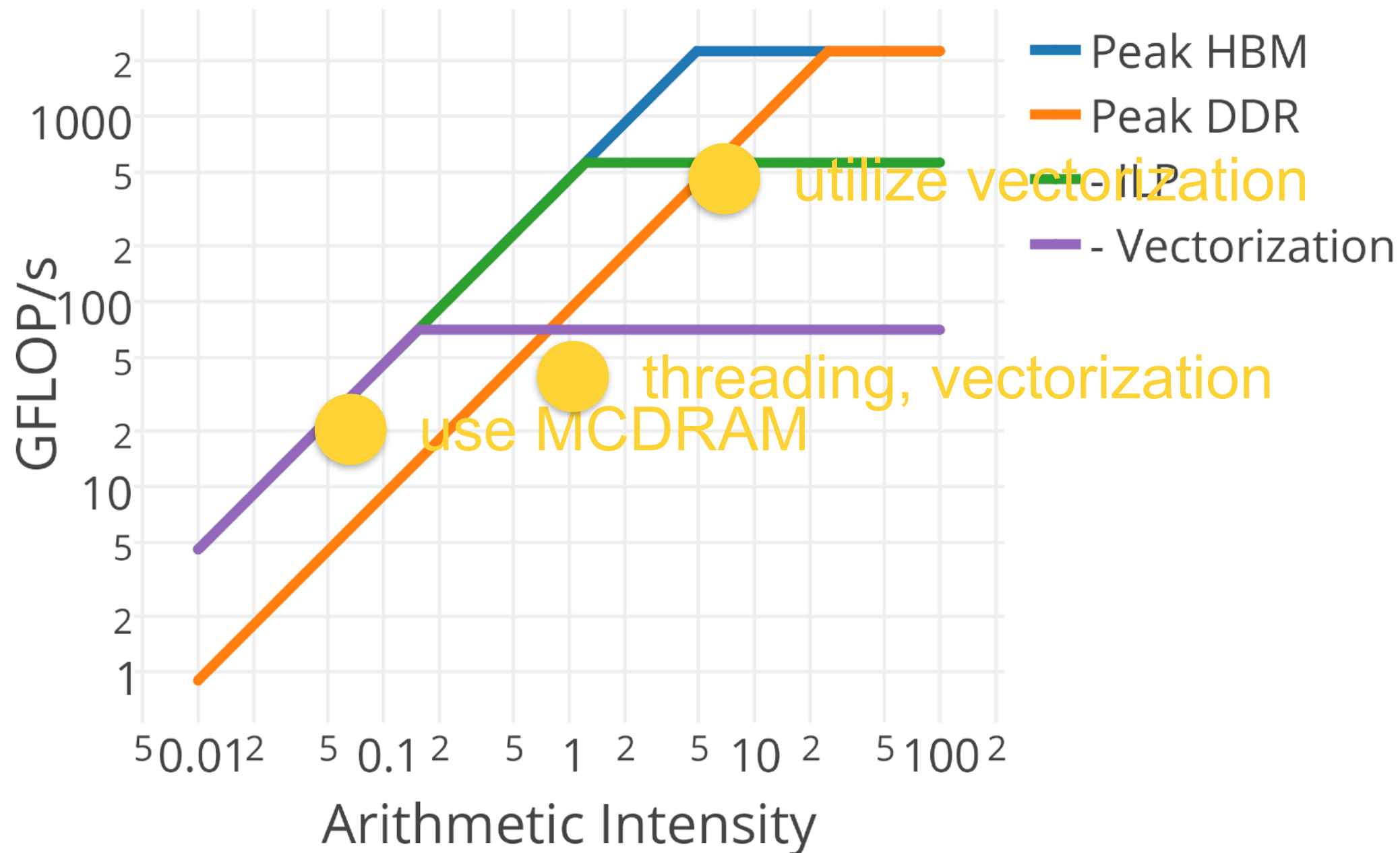
Example roofline



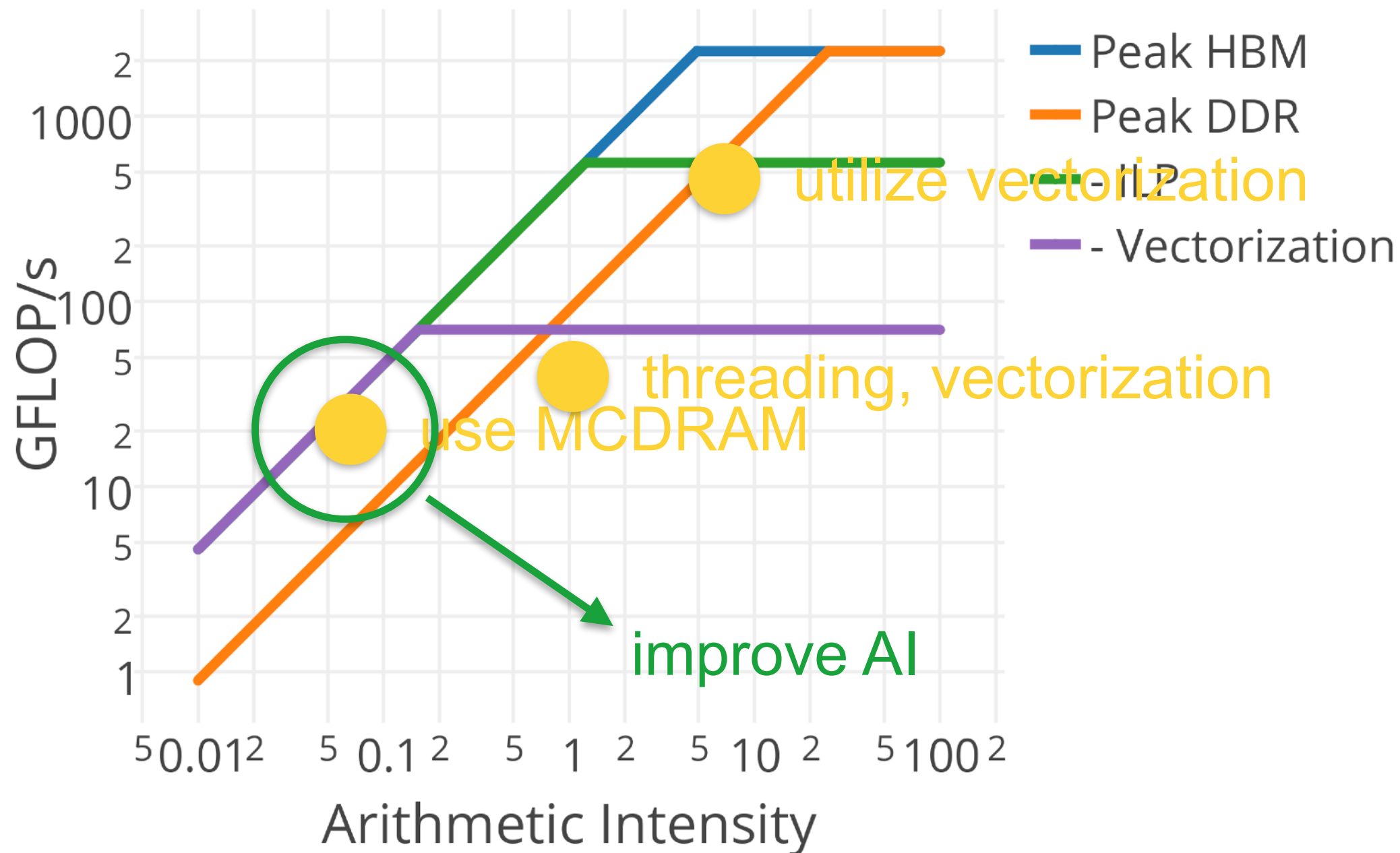
Example roofline



Example roofline



Example roofline



How to improve AI?



- definition of arithmetic intensity

$$AI = \frac{\#FLOPS}{\#BYTES}$$

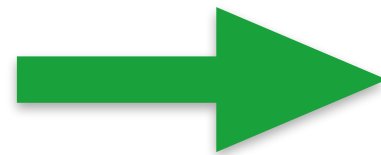
- two possibilities
 - number of flops \uparrow number of bytes \Rightarrow
(not possible/easy, choice of algorithm determines flops)
 - number of flops \Rightarrow number of bytes \downarrow
- reality: tradeoff between both

Create more work/thread



- **loop/kernel fusion:** improves cache re-use and reduce overhead

```
1  #pragma omp parallel for
2  for(unsigned int i=0; i<N; ++i){
3      r[i] = 0.;
4      for(unsigned int j=0; j<N; ++j){
5          r[i] += A[j+i*N]*x[j];
6      }
7  }
8
9  #pragma omp parallel for
10 for(unsigned int i=0; i<N; ++i){
11     r[i] = b[i] - r[i];
12 }
13
14 alpha=0.;
15 #pragma omp parallel for reduction(+:alpha)
16 for(unsigned int i=0; i<N; ++i){
17     alpha += r[i]*r[i];
18 }
```



```
22 alpha=0.;
23 #pragma omp parallel for reduction(+:alpha)
24 for(unsigned int i=0; i<N; ++i){
25     r[i] = 0.;
26     for(unsigned int j=0; j<N; ++j){
27         r[i] += A[j+i*N]*x[j];
28     }
29     r[i] = b[i] - r[i];
30     alpha += r[i]*r[i];
31 }
```

- **collapse nested loops:**

```
1  #pragma omp parallel for simd collapse(3)
2  for(unsigned int z=0; z<Nz; z++){
3      for(unsigned int y=0; y<Ny; y++){
4          for(unsigned int x=0; x<Nx; x++){
5              ...
6          }
7      }
8  }
```

- **rearrange data structures:** *move OpenMP out* (coarse grain)

Loop transformations I



- **loop tiling:** improves cache re-use and can significantly improve performance

```
1  !$omp parallel do collapse(2) firstprivate(n_jr, n_ir) --
2  !$omp private(jr, ir) --
3  DO jr = 1, n_jr --
4  > DO ir = 1, n_ir --
5  > > c(ir, jr) = a(ir, jr) * b(ir) --
6  > ENDDO --
7  ENDDO --
8  !$omp end parallel do --
```

Loop transformations I



- **loop tiling:** improves cache re-use and can significantly improve performance

```
12  nblock=2048
13  n_irt = n_ir / nblock
14  if (mod(n_ir, nblock) .ne. 0) n_irt = n_irt + 1
15  !$omp parallel do collapse(2) firstprivate(n_jr, n_ir, n_irt, nblock)
16  !$omp private (ir_start, ir_end, jr, ir)
17  DO irt = 1, n_irt
18  > DO jr = 1, n_jr
19  > > ir_start = (irt - 1) * nblock + 1
20  > > ir_end = min(ir_start + nblock - 1, n_ir)
21  > > DO ir = ir_start, ir_end
22  > > > c(ir, jr) = a(ir, jr) * b(ir)
23  > > ENDDO
24  > ENDDO
25  ENDDO
26  !$omp end parallel do
```

- especially relevant on KNL because of missing L3
- blocking to **shared L2 (512KiB)** usually good
- was my transformation successful? check L1, L2 miss rates, e.g. in VTune

Loop transformations II



- **short loop unrolling:** helps the compiler vectorizing the *right* loops

```
1  #pragma omp parallel for collapse(3)
2  for(unsigned int z=0; z<Nz; z++){
3      for(unsigned int y=0; y<Ny; y++){
4          for(unsigned int x=0; x<Nx; x++){
5             
6              vnorm(x,y,z) = 0.;
7              for(unsigned int d=0; d<3; d++){
8                  vnorm(x,y,z) += a(x,y,z)[d] * a(x,y,z)[d];
9              }
10         }
11     }
12 }
```

Loop transformations II



- **short loop unrolling:** helps the compiler vectorizing the *right* loops

```
16 #pragma omp parallel for simd collapse(3)
17 ▼ for(unsigned int z=0; z<Nz; z++){
18 ▼   ▶ for(unsigned int y=0; y<Ny; y++){
19 ▼   ▶   ▶ for(unsigned int x=0; x<Nx; x++){
20   ▶   ▶   ▶   -
21   ▶   ▶   ▶   vnorm(x,y,z) = a(x,y,z)[0] * a(x,y,z)[0]
22   ▶   ▶   ▶   ▶   ▶   ▶   + a(x,y,z)[1] * a(x,y,z)[1]
23   ▶   ▶   ▶   ▶   ▶   ▶   + a(x,y,z)[2] * a(x,y,z)[2];
24 ▲   ▶   ▶   }
25 ▲   ▶   }
26 ▲ }
```

- **unrolling pragmas** are helpful too
- check compiler optimization reports
- use Intel Advisor

- align (and pad) data to 64bit words to improve prefetching
- can be done easily in major programming languages
 - FORTRAN: `-align array64byte`
(ifort, gfortran does it automagically)
 - C/C++: `aligned_alloc(64, <size>),`
`__attribute__((aligned(64))), __declspec(align(64))`
 - C++ *trick*: overload new operator
- advanced: manually pad data if array extents are power of 2 to minimize cache associativity conflicts

Make use of ISA



- help the compiler to generate efficient intrinsics

```
1  for (int n = 0; n < nc; n++) {
2      for (int k = lo[2]; k <= hi[2]; ++k) {
3          for (int j = lo[1]; j <= hi[1]; ++j) {
4              #pragma omp simd
5                  for (int i = lo[0]; i <= hi[0]; i++) {
6                      if ((lo[0] + j + k + rb) % 2 != 0) continue;
7
8                      //BC terms
9                      Real cf0 = ( (i==blo[0]) && (m0(IntVect(blo[0]-1,j,k))>0) ? f0(IntVect(blo[0],j,k)) : 0.);
10                     Real cf1 = ( (j==blo[1]) && (m1(IntVect(i,blo[1]-1,k))>0) ? f1(IntVect(i,blo[1],k)) : 0.);
11                     Real cf2 = ( (k==blo[2]) && (m2(IntVect(i,j,blo[2]-1))>0) ? f2(IntVect(i,j,blo[2])) : 0.);
12                     Real cf3 = ( (i==bhi[0]) && (m3(IntVect(bhi[0]+1,j,k))>0) ? f3(IntVect(bhi[0],j,k)) : 0.);
13                     Real cf4 = ( (j==bhi[1]) && (m4(IntVect(i,bhi[1]+1,k))>0) ? f4(IntVect(i,bhi[1],k)) : 0.);
14                     Real cf5 = ( (k==bhi[2]) && (m5(IntVect(i,j,bhi[2]+1))>0) ? f5(IntVect(i,j,bhi[2])) : 0.);
15
16                     //assign ORA constants
17                     double gamma = alpha * a(IntVect(i,j,k))
18                         + dhx * (bX(IntVect(i,j,k)) + bX(IntVect(i+1,j,k)))
19                         + dhy * (bY(IntVect(i,j,k)) + bY(IntVect(i,j+1,k)))
20                         + dhz * (bZ(IntVect(i,j,k)) + bZ(IntVect(i,j,k+1)));
21
22                     double g_m_d = gamma
23                         - dhx * (bX(IntVect(i,j,k))*cf0 + bX(IntVect(i+1,j,k))*cf3)
24                         - dhy * (bY(IntVect(i,j,k))*cf1 + bY(IntVect(i,j+1,k))*cf4)
25                         - dhz * (bZ(IntVect(i,j,k))*cf2 + bZ(IntVect(i,j,k+1))*cf5);
26
27                     double rho = dhx * (bX(IntVect(i,j,k))*phi(IntVect(i-1,j,k),n) + bX(IntVect(i+1,j,k))*phi(IntVect(i+1,j,k),n))
28                         + dhy * (bY(IntVect(i,j,k))*phi(IntVect(i,j-1,k),n) + bY(IntVect(i,j+1,k))*phi(IntVect(i,j+1,k),n))
29                         + dhz * (bZ(IntVect(i,j,k))*phi(IntVect(i,j,k-1),n) + bZ(IntVect(i,j,k+1))*phi(IntVect(i,j,k+1),n));
30
31                     double res = rhs(IntVect(i,j,k),n) - gamma * phi(IntVect(i,j,k),n) + rho;
32                     phi(IntVect(i,j,k),n) += omega/g_m_d * res;
33                 }
34             }
35         }
36     }
```

runtime example for app with kernel: 1.2 sec

Make use of ISA



- help the compiler to generate efficient intrinsics

```
1  for (int n = 0; n < nc; n++) {
2      for (int k = lo[2]; k <= hi[2]; ++k) {
3          for (int j = lo[1]; j <= hi[1]; ++j) {
4              #pragma omp simd
5                  for (int i = lo[0]; i <= hi[0]; i++) {
6                      if ((lo[0] + j + k + rb) % 2 != 0) continue;
7
8                      //BC terms
9                      Real cf0 = ( (i==blo[0]) && (m0(IntVect(blo[0]-1,j,k))>0) ? f0(IntVect(blo[0],j,k)) : 0.);
10                     Real cf1 = ( (j==blo[1]) && (m1(IntVect(i,blo[1]-1,k))>0) ? f1(IntVect(i,blo[1],k)) : 0.);
11                     Real cf2 = ( (k==blo[2]) && (m2(IntVect(i,j,blo[2]-1))>0) ? f2(IntVect(i,j,blo[2])) : 0.);
12                     Real cf3 = ( (i==bhi[0]) && (m3(IntVect(bhi[0]+1,j,k))>0) ? f3(IntVect(bhi[0],j,k)) : 0.);
13                     Real cf4 = ( (j==bhi[1]) && (m4(IntVect(i,bhi[1]+1,k))>0) ? f4(IntVect(i,bhi[1],k)) : 0.);
14                     Real cf5 = ( (k==bhi[2]) && (m5(IntVect(i,j,bhi[2]+1))>0) ? f5(IntVect(i,j,bhi[2])) : 0.);
15
16                     //assign ORA constants
17                     double gamma = alpha * a(IntVect(i,j,k))
18                         + dhx * (bX(IntVect(i,j,k)) + bX(IntVect(i+1,j,k)))
19                         + dhy * (bY(IntVect(i,j,k)) + bY(IntVect(i,j+1,k)))
20                         + dhz * (bZ(IntVect(i,j,k)) + bZ(IntVect(i,j,k+1)));
21
22                     double g_m_d = gamma
23                         - dhx * (bX(IntVect(i,j,k))*cf0 + bX(IntVect(i+1,j,k))*cf3)
24                         - dhy * (bY(IntVect(i,j,k))*cf1 + bY(IntVect(i,j+1,k))*cf4)
25                         - dhz * (bZ(IntVect(i,j,k))*cf2 + bZ(IntVect(i,j,k+1))*cf5);
26
27                     double rho = dhx * (bX(IntVect(i,j,k))*phi(IntVect(i-1,j,k),n) + bX(IntVect(i+1,j,k))*phi(IntVect(i+1,j,k),n))
28                         + dhy * (bY(IntVect(i,j,k))*phi(IntVect(i,j-1,k),n) + bY(IntVect(i,j+1,k))*phi(IntVect(i,j+1,k),n))
29                         + dhz * (bZ(IntVect(i,j,k))*phi(IntVect(i,j,k-1),n) + bZ(IntVect(i,j,k+1))*phi(IntVect(i,j,k+1),n));
30
31                     double res = rhs(IntVect(i,j,k),n) - gamma * phi(IntVect(i,j,k),n) + rho;
32                     phi(IntVect(i,j,k),n) += omega/g_m_d * res;
33                 }
34             }
35         }
36     }
```

if condition inside loop

runtime example for app with kernel: 1.2 sec

Make use of ISA



- help the compiler to generate efficient intrinsics

```

1  for (int n = 0; n < nc; n++) {
2      for (int k = lo[2]; k <= hi[2]; ++k) {
3          for (int j = lo[1]; j <= hi[1]; ++j) {
4              int ioff = (lo[0] + j + k + rb) % 2;
5              #pragma omp simd
6              for (int i = lo[0] + ioff; i <= hi[0]; i += 2) {
7                  //BC terms
8                  Real cf0 = ((i == blo[0]) && (m0(IntVect(blo[0]-1,j,k)) > 0) ? f0(IntVect(blo[0],j,k)) : 0.);
9                  Real cf1 = ((j == blo[1]) && (m1(IntVect(i,blo[1]-1,k)) > 0) ? f1(IntVect(i,blo[1],k)) : 0.);
10                 Real cf2 = ((k == blo[2]) && (m2(IntVect(i,j,blo[2]-1)) > 0) ? f2(IntVect(i,j,blo[2])) : 0.);
11                 Real cf3 = ((i == bhi[0]) && (m3(IntVect(bhi[0]+1,j,k)) > 0) ? f3(IntVect(bhi[0],j,k)) : 0.);
12                 Real cf4 = ((j == bhi[1]) && (m4(IntVect(i,bhi[1]+1,k)) > 0) ? f4(IntVect(i,bhi[1],k)) : 0.);
13                 Real cf5 = ((k == bhi[2]) && (m5(IntVect(i,j,bhi[2]+1)) > 0) ? f5(IntVect(i,j,bhi[2])) : 0.);
14
15                 //assign ORA constants
16                 double gamma = alpha * a(IntVect(i,j,k))
17                     + dhx * (bX(IntVect(i,j,k)) + bX(IntVect(i+1,j,k)))
18                     + dhy * (bY(IntVect(i,j,k)) + bY(IntVect(i,j+1,k)))
19                     + dhz * (bZ(IntVect(i,j,k)) + bZ(IntVect(i,j,k+1)));
20
21                 double g_m_d = gamma
22                     - dhx * (bX(IntVect(i,j,k))*cf0 + bX(IntVect(i+1,j,k))*cf3)
23                     - dhy * (bY(IntVect(i,j,k))*cf1 + bY(IntVect(i,j+1,k))*cf4)
24                     - dhz * (bZ(IntVect(i,j,k))*cf2 + bZ(IntVect(i,j,k+1))*cf5);
25
26                 double rho = dhx * (bX(IntVect(i,j,k))*phi(IntVect(i-1,j,k),n) + bX(IntVect(i+1,j,k))*phi(IntVect(i+1,j,k),n))
27                     + dhy * (bY(IntVect(i,j,k))*phi(IntVect(i,j-1,k),n) + bY(IntVect(i,j+1,k))*phi(IntVect(i,j+1,k),n))
28                     + dhz * (bZ(IntVect(i,j,k))*phi(IntVect(i,j,k-1),n) + bZ(IntVect(i,j,k+1))*phi(IntVect(i,j,k+1),n));
29
30                 double res = rhs(IntVect(i,j,k),n) - gamma * phi(IntVect(i,j,k),n) + rho;
31                 phi(IntVect(i,j,k),n) += omega/g_m_d * res;
32             }
33         }
34     }
35 }
36

```

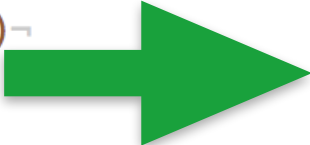
runtime example for app with kernel: 0.8 sec

1.5X SPEEDUP

Reduced precision math



- transcendental functions, square roots, etc. are expensive
- use `-fp-model fast=2 -no-prec-div` during compilation
- replace divisions by constants with multiplications with inverse

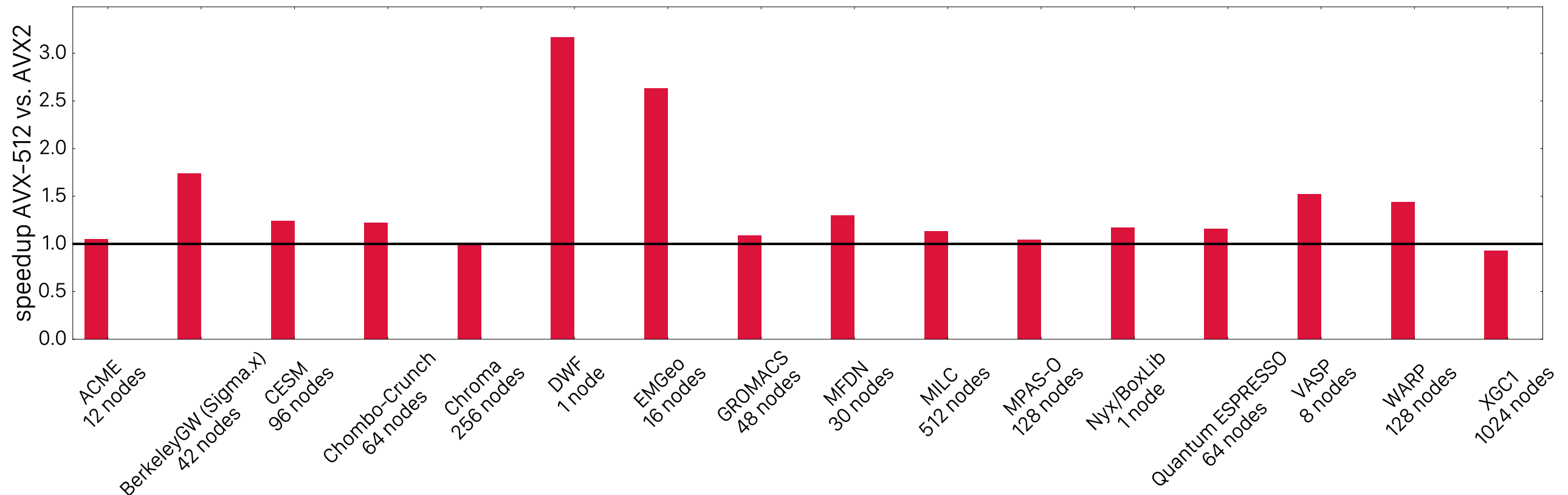


```
1 double norm=1.234;
2 #pragma omp parallel for firstprivate(norm)
3 for(unsigned int i=0; i<N; i++){
4     a[i]/=norm;
5 }

9 double invnorm=1./1.234;
10 #pragma omp parallel for firstprivate(invnorm)
11 for(unsigned int i=0; i<N; i++){
12     a[i]*=invnorm;
13 }
```

- do not expect too much: benefits usually only visible in heavily compute-bound code sections
- reduced precision might not always be acceptable

Benefits of AVX-512

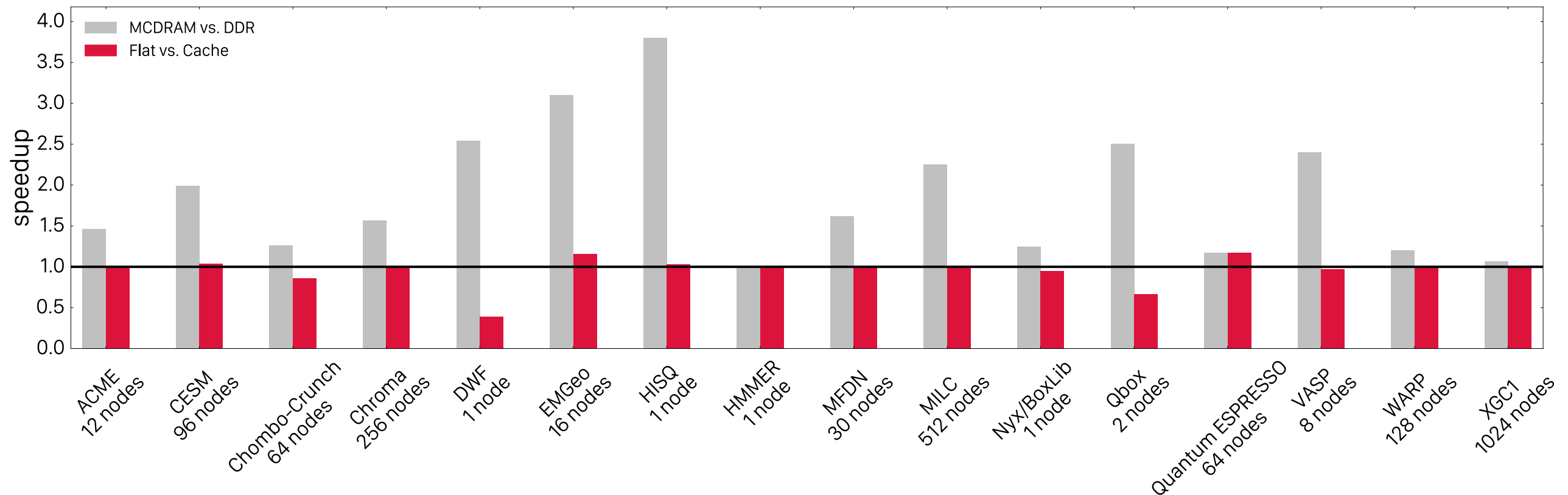


- median speedup: 1.2x
- benefits can be larger than 2x
(probably more efficient prefetching)
- automatically enabled when compiling for KNL architecture

Use MCDRAM



- always use 16GiB on-package memory (MCDRAM)



- **cache works well:** request with `-C knl,cache`
- **code fits into 16GiB:** request `-C knl,flat` and prepend executable with `numactl -m 1`

A note on heap allocation

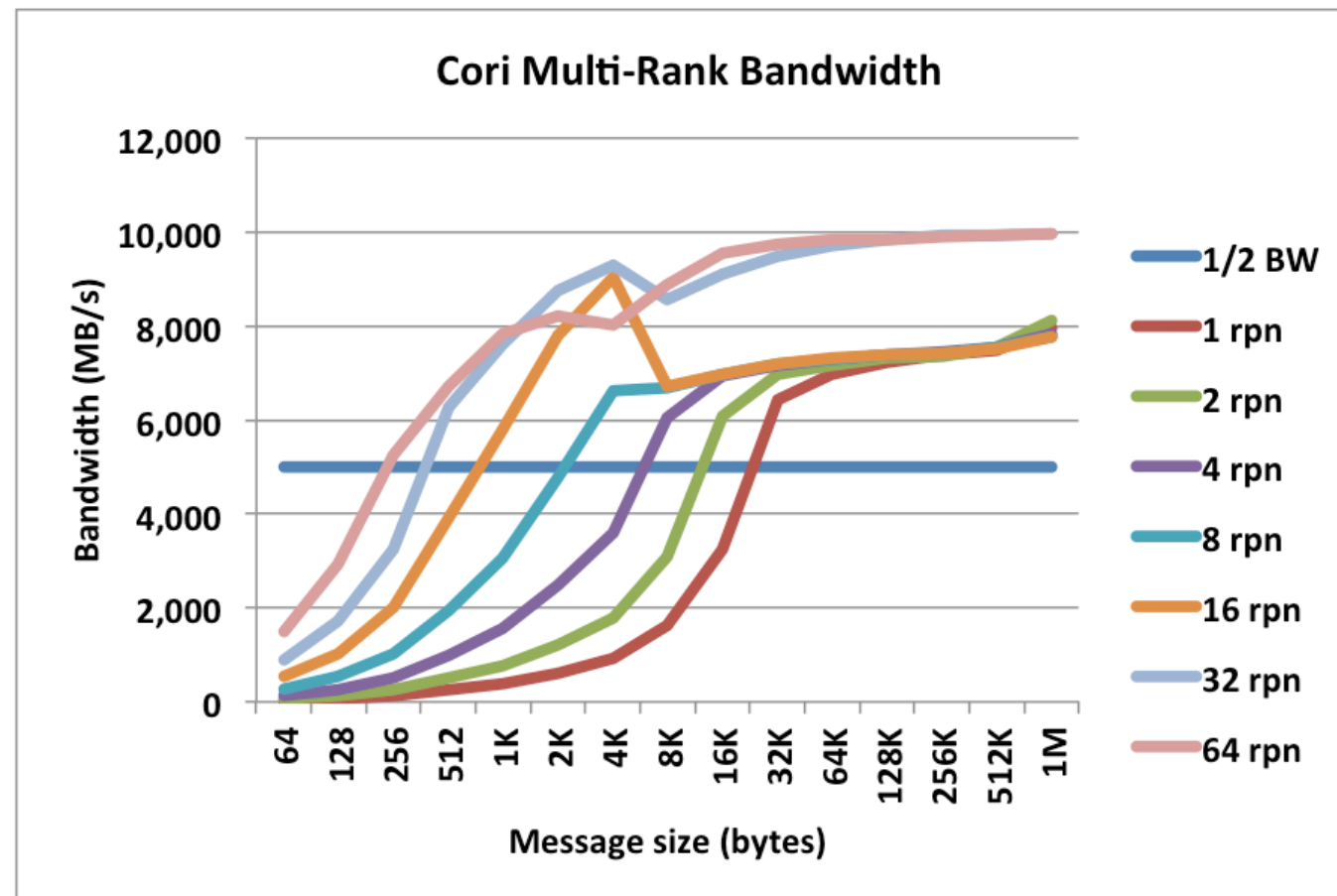


- KNL memory allocation is comparably slow
- **avoid allocating and de-allocating memory frequently**
 - remove allocations/deallocations in loop bodies or functions which are called many times
- too involved? **pool allocator libraries** (e.g. [Intel TBB scalable memory pools](#))
 - pros:
 - overloads new/malloc, no/minimal source code changes necessary
 - can give significant performance boost for certain codes
 - take care of thread-safety
 - cons:
 - memory footprint needs to be known/computed in advance
 - code might become less portable

Multi-node optimizations



- single KNL thread cannot saturate Aries injection rate



- use thread-level communication or **multiple MPI ranks per node**
- recommended: **>4 ranks per node**
- **dedicate cores to OS:** -S <ncores> in sbatch (ncores=2 good choice)

Hugepages, DMAPP and hardware AMO



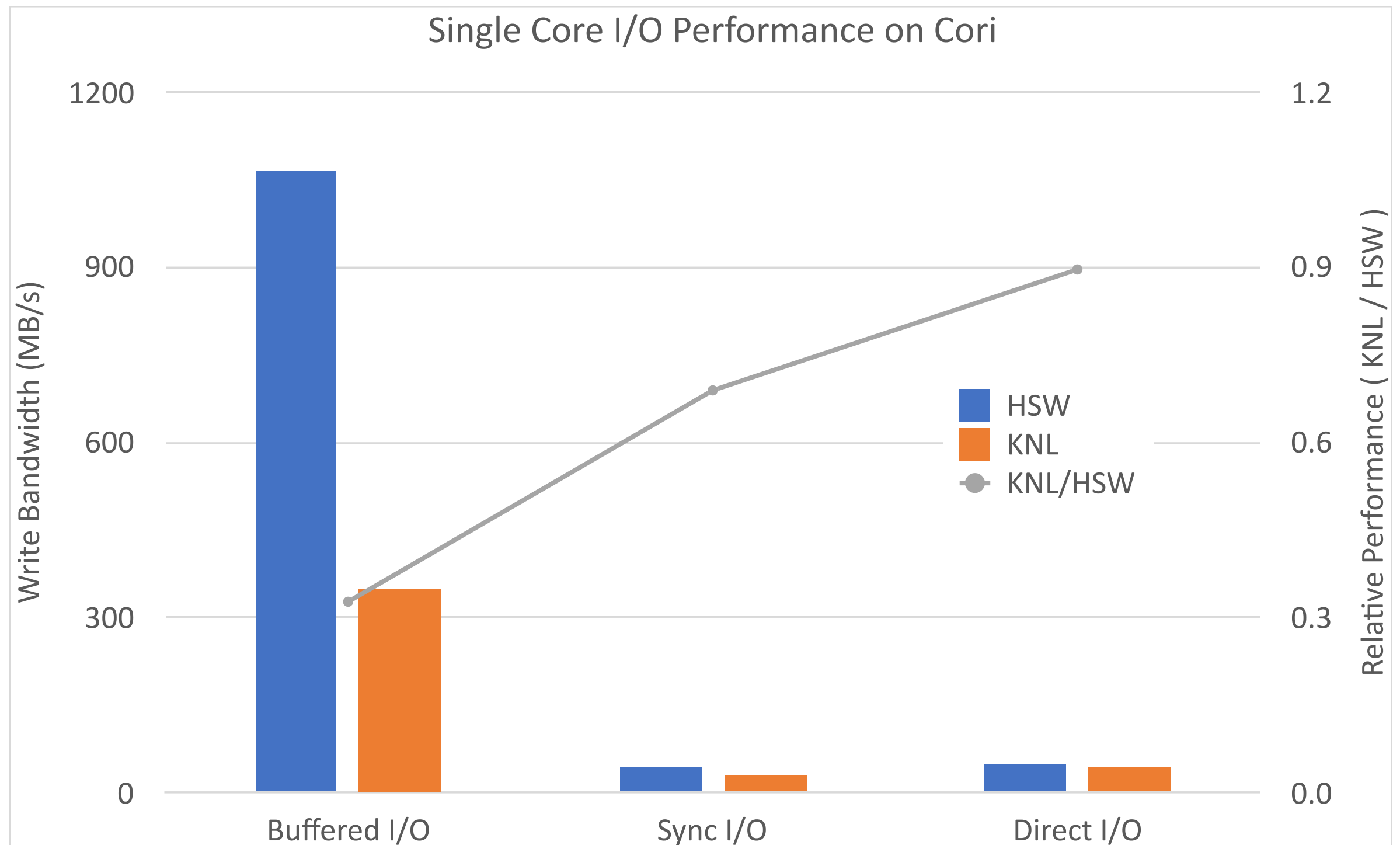
- **hugepages** can reduce Aries TLB misses
- load corresponding module at compile and runtime

```
[[tkurth@cori04 ~]$ module avail craype-hugepages
```

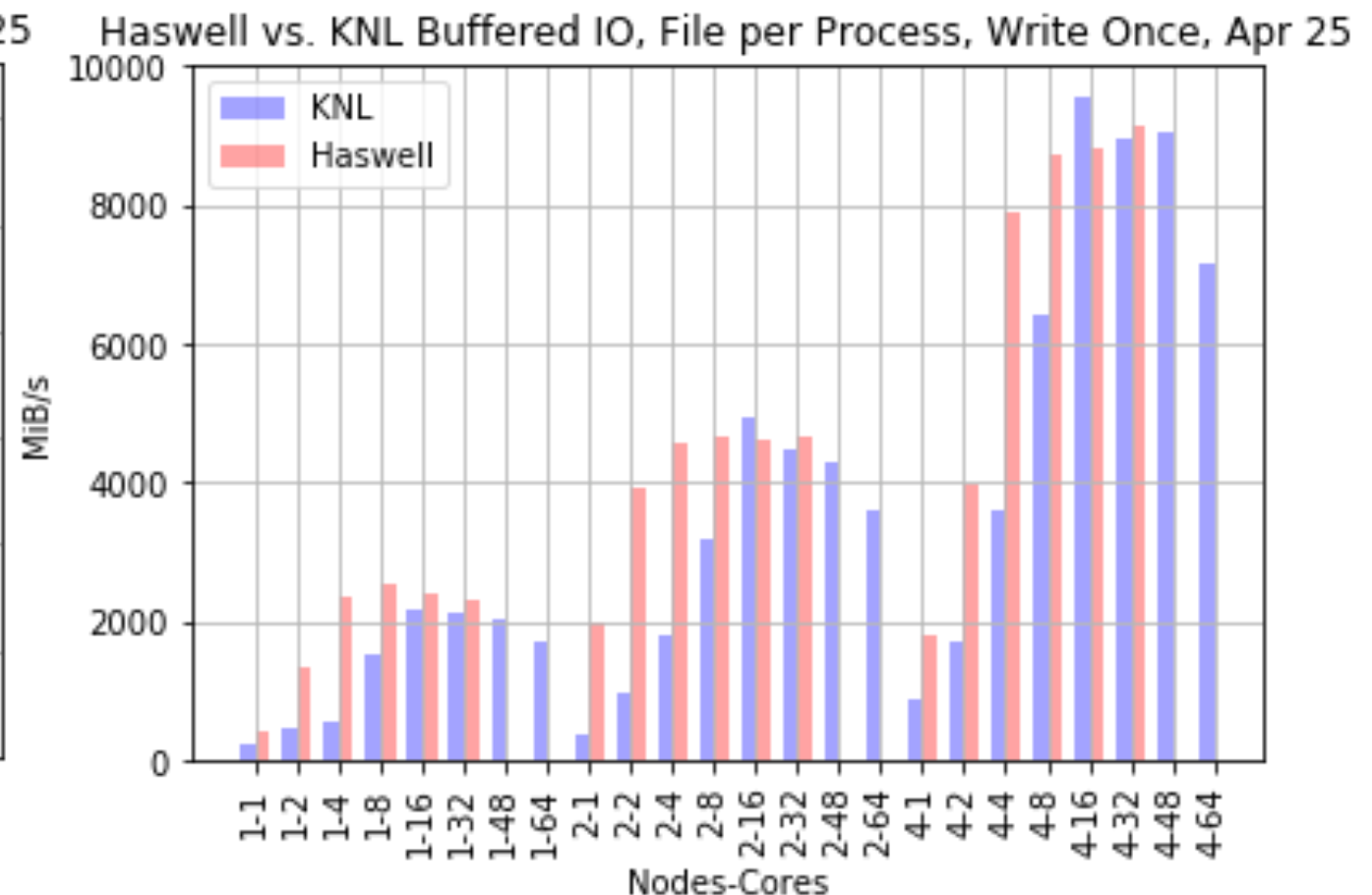
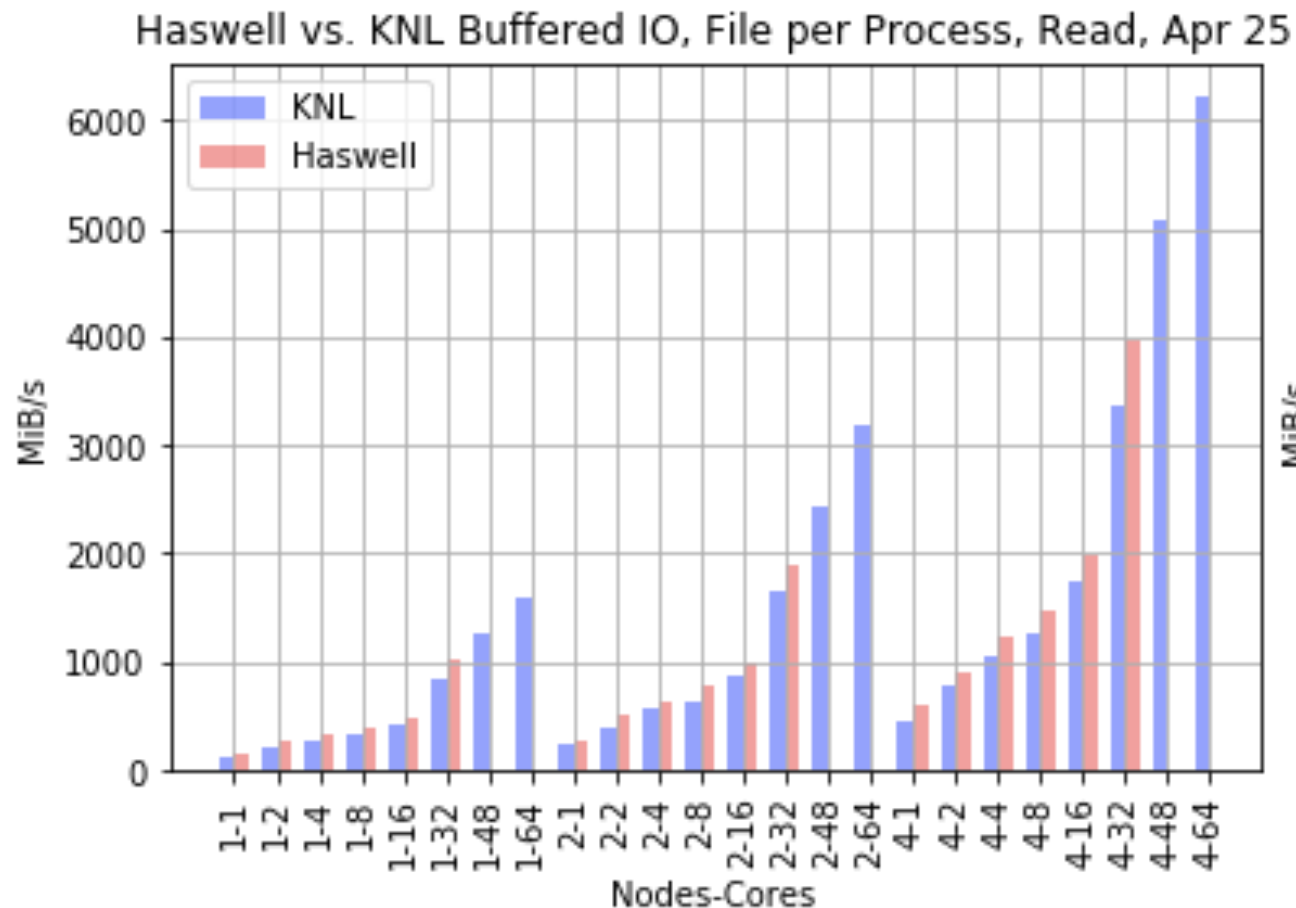
```
----- /opt/cray/pe/craype/2.5.7/modulefiles -----  
craype-hugepages128M  craype-hugepages256M  craype-hugepages32M  craype-hugepages512M  craype-hugepages8M  
craype-hugepages16M   craype-hugepages2M    craype-hugepages4M    craype-hugepages64M  
-----
```

- can use **different modules at compile and runtime**
- MPI-collective-heavy codes: **enable DMAPP** (add `-ldmapp`)
export MPICH_RMA_OVER_DMAPP=1
export MPICH_USE_DMAPP_COLL=1
export MPICH_NETWORK_BUFFER_COLL_OPT=1
- enable hardware AMO for MPI-3 RMA atomics
export MPICH_RMA_USE_NETWORK_AMO=1

Some notes on IO

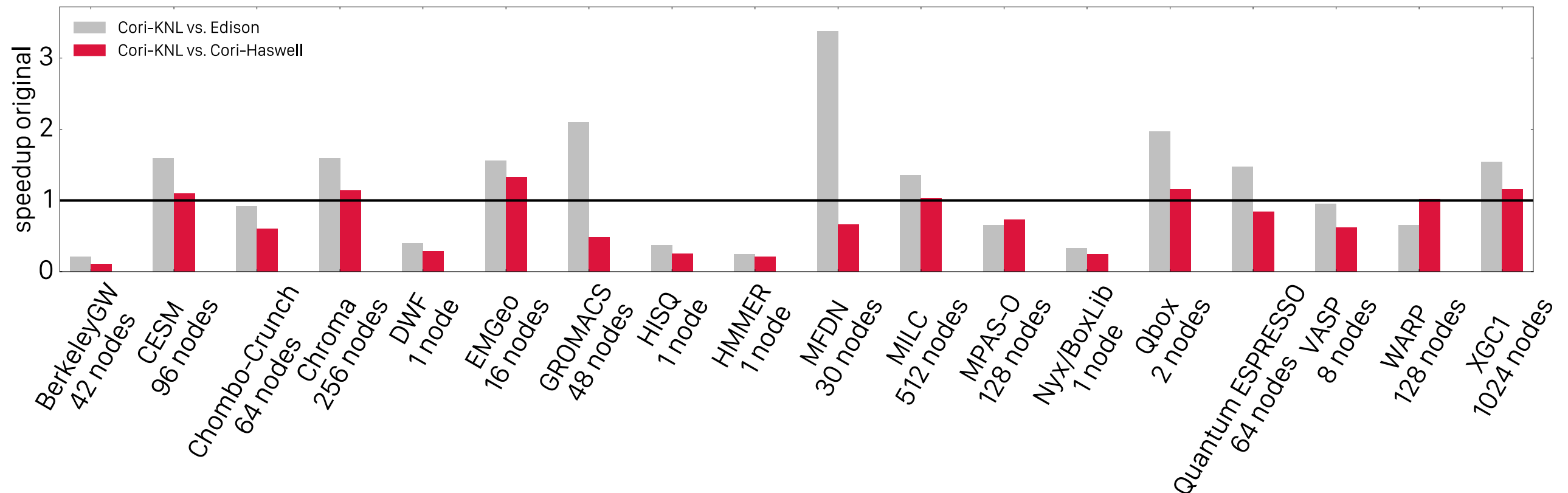


Use multiple processes



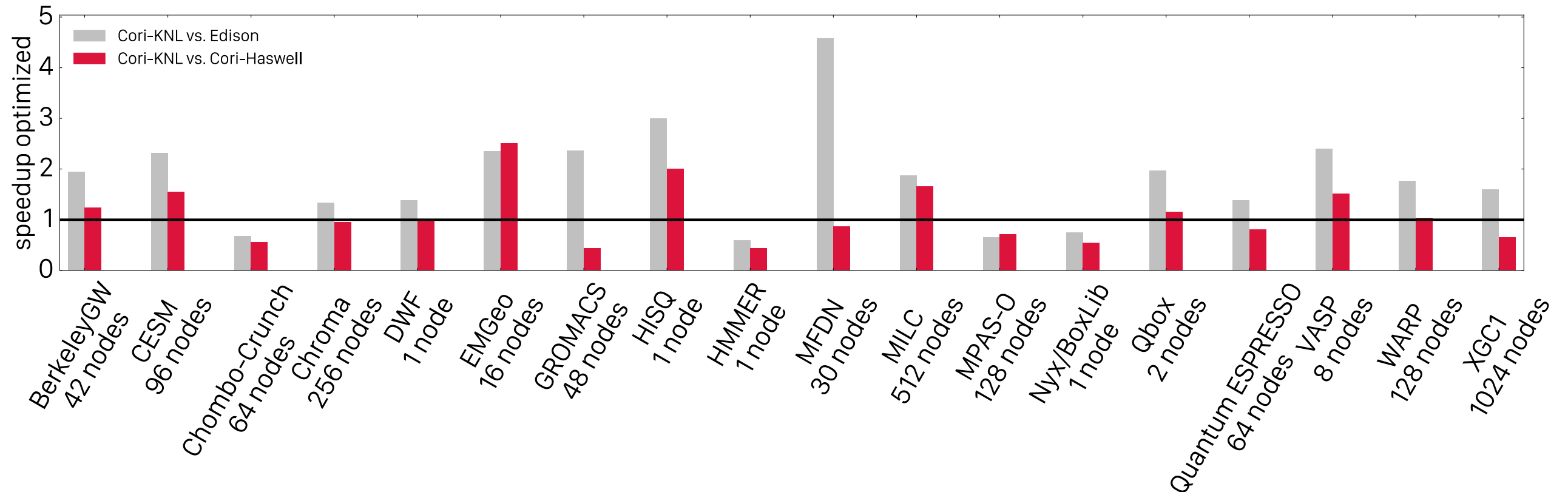
- use more processes (e.g. with MPIIO)
- unfortunately, no good threaded IO solutions available yet
- always: pool (write big chunks), reduce file operations (open, close)
- large files: [burst buffer](#)

Does it help?



- median speedup vs. Edison: 1.15x
- median speedup vs. Haswell: 0.70x

Does it help?



- median speedup vs. Edison: 1.8x
- median speedup vs. Haswell: 1.0x

- single node performance (go for that one first)
 - loop fusion and tiling
 - ensure good vectorization
 - use MCDRAM
- multi-node performance
 - hugepages
 - DMAPP
- IO performance
 - use multiple nodes, pool IO, reduce file operations to minimum

- [running jobs](#)
- [process/thread binding](#)
- [code profiling and tools](#)
- [measuring arithmetic intensity \(AI\)](#)
- [improving OpenMP scaling](#)
- [vectorization help](#)
- [how to use MCDRAM](#)
- [NESAP case studies](#)

NERSC

Thank you